Lesson 7 - Team Assignment

Remember that the View layer is responsible for the interaction with the end user. It retrieves inputs from the end user, determines and calls the appropriate Control Layer function (as needed) to fulfill the request, and then determines and displays the next view back to the end user.

In this assignment, you will work with your team member to create the View Layer classes for the **Start program**, **Main menu**, and **Getting help** end user stories for your game. You will be using selection statements, repetition statements, and `String` functions to create these views.

Step 1 - Create the Start Program view

Start by reading the end user story for the view your are implementing. Here is the **Start Program** end user story for the example program.

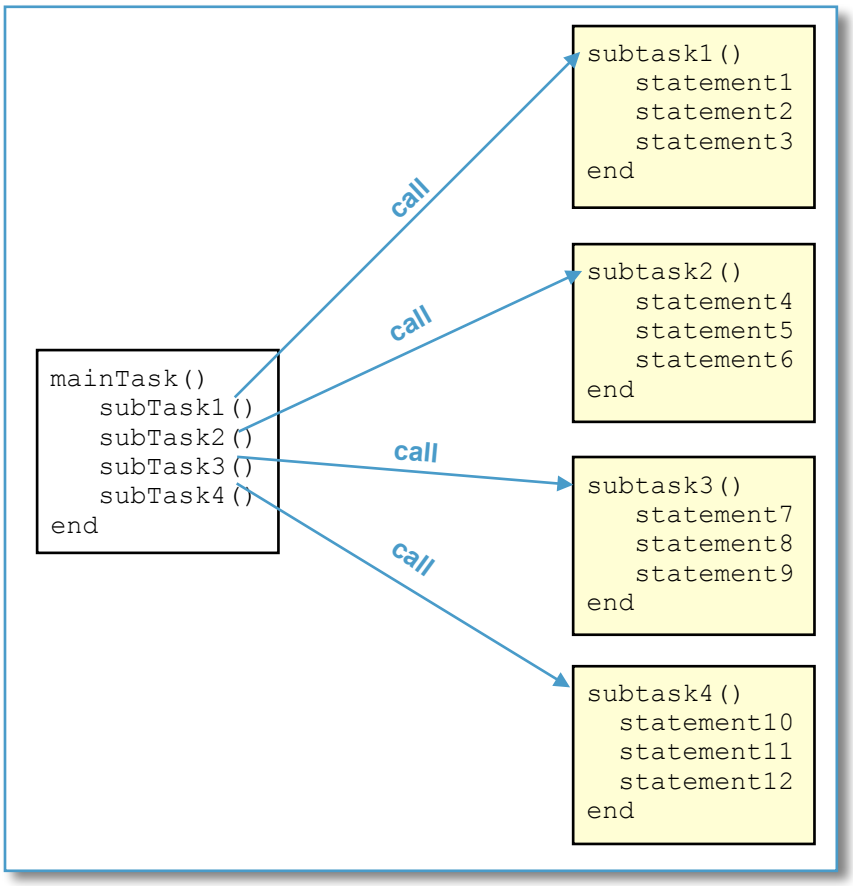
> ***Start program***
>
> The end user enters the command to start the program. The computer displays a banner screen with a short description of the game. The end user (player) is then prompted to enter their name. The user enters their name and the computer saves the name, displays a personalize welcome message and the Main Menu.

We will create a new class in the View Layer package called `StartProgramView` to implement the **Start program** view. This class will have a function called `displayStartProgramView()` that is responsible for displaying the view.

Using Divide and Conquer to solve complex problems

The algorithm for the `displayStartProgramView()` function is rather complex because it has to do a lot of things. It can get confusing to think about all of the code that needs to be written all at once. The **Divide and Conquer** problem solving strategy is often used in programming to solve complex problems. First, we break the algorithm down into it's most basic steps. Then we break down each of those basic steps into subtask and create separate functions for each of the subtask. The original function will then call each of the subtask functions in the correct order to perform the basic steps of the algorithm.

This approach allows us to first focus on the big picture and get the basic steps in the correct order without worrying about the nitty, gritty details of how to implement each of the more detailed subtasks. Later on we will implement each of the functions for each subtask as a separate individual problems. This concept not only makes it easier to solve complex problems, but it also makes our code much more readable, easy to debug and change. This concept of **Divide and Conquer** is illustrated below.



We will use this **Divide and Conquer** problem solving strategy to develop the `StartProgramView` class and it's `displayStartProgramView()` function.

> ***Start program***
>
> The end user enters the command to start the program. The computer displays a banner screen with a short description of the game. The player is prompted for and enters their name. The user enters their name and the computer creates a new Player and then displays a personalize welcome message for the player. The Main Menu is then displayed. An error message is displayed if an invalid name is entered and the player is prompted to reenter a valid name or quit.

The `displayStartProgramView()` function is responsible for implementing the end user request cycle. This function must implement the following steps:

1. Printing the banner page with a short description of the game.
2. Prompting for and getting the players name
3. Calling a control function to create a player object
4. If the player is created successfully, display a welcome message and the main menu; else, display an error message and repeat steps 2 -4.

Several of these steps can be quite complicated in themselves. Using Divide and Conquer we can break these steps up into separate subtask and create separate functions for the more complicated task. We will create six separate functions that perform the following subtask:

- Displays the banner page.
- Display Start Program View
- Prompt for and gets the players name
- Perform the desire action
- A Control function to create a new Player object.
- Display a custom welcome message and the Main Menu.

Create a View Layer package and the StartProgramView class

Start by creating a new package for the View Layer in your project. Call the package:

`citbyui.cit260.`*yourprojectName*`.view`

Now create a new class in your View Layer package to implement the view for the **Start Program** end user story called `StartProgramView`. This class will contain all of the functions that deal with input and output in the **Start Program** end user story.

Implementing the default constructor function

The banner page will be displayed as the first thing when a new object instance of the `StartProgramView` class is created. The default construction function for the class is called automatically called when a new object is instantiated. The constructor function is a good place to set default values for class instance variables and to perform any other actions that are associated with the creation of the object. Here is the algorithm for the `StartProgramView()` constructor function.

```
StartProgramView(): void
BEGIN
  promptMessage = "Please enter your name:"
  display the banner page
END
```

The constructor function always has the same name as the name of the class and always has a return type of `void` (returns nothing). This function will assign a value to the promptMessage class instance variable. This is the default message that will be displayed each time the end user is prompted to enter input. Next, we will display the banner page. The banner page is quite lengthy and as a result we will create a special function to display the banner page.

Create of the `StartProgramView()` constructor function in the `StartProgramView` class and then type the algorithm in comments as follows.

```
public class StartProgramView {

    public StartProgramView() {
        // promptMessage = "Please enter your name"
        // display the banner when view is created
    }
```

Start by implementing the first line of the algorithm as shown.

```
public class StartProgramView {

    private String promptMessage;

    public StartProgramView() {

        this.promptMessage = "\nPlease enter your name: ";
        // display the banner when view is created

    }
```

The `promptMessage` variable will used in another function in this class so will need to make this a class instance variable. Then we assign the literal text string, "*Please enter your name:*" to the variable. Notice that the `promptMessage` variable is qualified with the keyword `this`. The `this` keyword tells Java to look for a class instance variable in this "this" class called `promptMessage`.

Implement the next line of the algorithm by calling a separate function called `displayBanner()`. Again, we qualified the function with the `this` keyword because it is to be found in this same class.

```
public class StartProgramView {

    private String promptMessage;

    public StartProgramView() {

        this.promptMessage = "\nPlease enter your name: ";
        // display the banner when view is created
        this.displayBanner();

    }
```

Notice that the line is underlined in red indicating an error because this function has not been created yet. Implement the `displayBanner()` function by clicking on the lightbulb next to the line number and selecting the following hint.

```
12    public class StartProgramView {
13
15        private String promptMessage;
16
16        public StartProgramView() {
17
18            this.promptMessage = "\nPlease enter your name: ";
19            // display the banner when view is created
            this.displayBanner();
21
22
```

Select

Create method "displayBanner()" in citbyui.cit260.curiousworkmanship.view.StartProgramView

This `displayBanner()` function will automatically be created for you at the bottom of the class.

```
public class StartProgramView {

    private String promptMessage;

    public StartProgramView() {

        this.promptMessage = "\nPlease enter your name: ";
        // display the banner when view is created
        this.displayBanner();

    }

    private void displayBanner() {
        throw new UnsupportedOperationException("Not supported yet.");
    }

}
```

Implement the `displayBanner()` *function*

The implementation of `displayBanner()` function is straight forward. It prints the display banner with a short description of the program to the screen console. Delete the `throw UnsupportedOperationException` statement, and create a `displayBanner()` function for your game similar to the one below.

```java
public void displayBanner() {

    System.out.println(
            "\n*************************************************"
          + "\n*                                               *"
          + "\n* This is the game of Curious Workmanship       *"
          + "\n* In this game you will help Nephi build a      *"
          + "\n* shiop of curious workmanship to travel to     *"
          + "\n* the promised land.                            *"
          + "\n*                                               *"
          + "\n* You and your family will need to first        *"
          + "\n* plan for your trip determining and            *"
          + "\n* and estimating the amount of resources        *"
          + "\n* needed for the trip. Then you will have       *"
          + "\n* to go out and search for and harvest          *"
          + "\n* the needed resources and deliver them         *"
          + "\n* to the warehouse where you will store         *"
          + "\n* then until the ship is completed. Then        *"
          + "\n* You will also need to build the ship,         *"
          + "\n* load the ship and then set sail for           *"
          + "\n* the promised land. You will first need        *"
          + "\n* to find the resources and manufacture the     *"
          + "\n* tools need to build the ship.                 *"
          + "\n                                                 *"
          + "\n* Good luck and have fun in this adventure.     *"
          + "\n*                                               *"
          + "\n*************************************************"
          );
}
```

Good programming practice:

A good programming practice is to write a little bit of code and then test it. Write some more code and test it. Using this approach allows us to isolate and find bugs easily. If a run time error occurs, it is most likely in the new code just added. If no error occurs, then you can be fairly confident that the new code added is bug free.

Test the code that you have written so far and make sure that it is working correctly. Execution of a Java program always starts in the class containing the `main()` function. We need to modify this function to call the `displayStartProgramView()` function.

Open the class that contains the `main()` function in your project. The `CuriousWorkmanship` class contains the `main()` function in the example program. Delete any existing code currently in the function and then add the following lines of code to the function. Don't worry! The code in the `main()` function is no longer needed. This code was written initially to test that the Model Layer classes to see that they were working correctly. Now we need to replace it with the real code that will display the first view of your program.

```
public static void main(String[] args) {

    // create StartProgramViewOrig and display the start program view
    StartProgramView startProgramView = new StartProgramView();
    startProgramView.displayStartProgramView();
}
```

The first statement in main function creates a new object instance of the `StartProgramView` class. The `new` keyword tells the computer to create a new object instance. This is followed by a call to the default constructor function of the `StartProgramView` class. Every class in java has a default constructor function. The name of a constructor function always matches the name of the class. Java looks at the name of the constructor function to locate the class, and then goes to that class and calls it's default constructor function. The new object instance is then assigned to the `startProgramView` variable whose data type matches that of the `StartProgramView` class. Remember that Java is a strongly typed language. The data type of the variable on the left of the equal sign must always be consistent with the data type of the object being assigned to it on the right.

Notice the lightbulb on the first line of code that you added. This indicates that there is an error on this line.

```
21      public static void main(String[] args) {
22
23          // create StartProgramViewOrig and display the start program view
            StartProgramView startProgramView = new StartProgramView();
25          startProgramView.displayStartProgramView();
26      }
```

Hover the pointer over the light bulb to see what the error is. The `cannot find symbol` error occurs again because Java can not find the name class.

```
16                                  player = null;
17
18
19
20
21                                  n(String[] args) {
22
23                                  gramViewOrig and display the start program view
                    StartProgramView startProgramView = new StartProgramView();
25                  startProgramView.displayStartProgramView();
26              }
```

cannot find symbol
  symbol: class StartProgramView
  location: class CuriousWorkmanship

cannot find symbol
  symbol: class StartProgramView
  location: class CuriousWorkmanship

(Alt-Enter shows hints)

In this case, we have defined this class in another file but not imported the class into this source file.

Select the lightbulb on the line where the error occurred and one or more suggestions are shown to fix the problem. Select the suggestion to import the `StartProgramView` class.

```
29              // create StartProgramView and start the program
                StartProgramView startProgramView = new StartProgramView();
31
32
33
34              public static Game getCurrentGame() {
```

Add import for citbyui.cit260.curiousworkmanship.view.StartProgramView
Create class "StartProgramView" in package curiousworkmanship
Create class "StartProgramView" in curiousworkmanship.CuriousWorkmanship

The next statement in the `main()` function calls the `displayStartProgramView()` function. The name of the function is underlined in red indicating an error. Hover over the lightbulb again to find the cause of this error. Java can not find the `displayStartProgramView()` function in the `StartProgramView` class.

```
21
22                                  {
23
24                                  display the start program view
25                                  new StartProgramView();
                    startProgramView.displayStartProgramView();
27              }
```

cannot find symbol
  symbol: method displayStartProgramView()
  location: variable startProgramView of type StartProgramView
---
(Alt-Enter shows hints)

We need to create this function. We can automatically create a stub function for this function. A stub function contain little or no functionality at this point. It's purpose is to allow us to run our program a test whether the function gets called or not.

Click on the lightbulb on the line number at the left and select the suggestion to create the `displayStartProgramView()` function.

```
22      public static void main(String[] args) {
23
24          // create StartProgramViewOrig and display the start program view
25          StartProgramView startProgramView = new StartProgramView();
26          startProgramView.displayStartProgramView();
27
28
```

Create method "displayStartProgramView()" in citbyui.cit260.curiousworkmanship.view.StartProgramView

Open up the StartProgramView class and locate the `displayStartProgramView()` function. Notice the throw new `UnsupportedOperationException` statement.

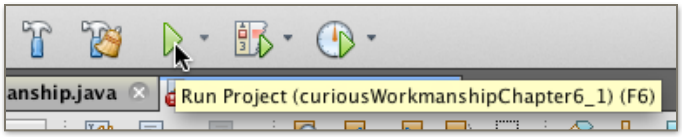
```
public class StartProgramView {

    public void displayStartProgramView() {
        throw new UnsupportedOperationException("Not supported yet.");
    }
```

This statement will force your program to fail when this function is called and has been inserted to force you as a programer to implement this function. For now we want to replace this with a statement that just prints a message indicating that the function was called. This will allow us to run and test our program to see if it is working.

```
/**
 * displays the start program view
 */
public void displayStartProgramView() {
    System.out.println("\n*** displayStartProgram() function called ***");
}
```

Now run and test your program by clicking on the run button at the top of the window.



The banner should now display with message indicating that you called the `displayStartProgramView()` function in the Output tab at the bottom of the screen.

```
**********************************************
*                                            *
* This is the game of Curious Workmanship    *
* In this game you will help Nephi build a   *
* shiop of curious workmanship to travel to  *
* the promised land.                         *
*                                            *
* You and your family will need to first     *
* plan for your trip determining and         *
* and estimating the amount of resources     *
* needed for the trip. Then you will have    *
* to go out and search for and harvest       *
* the needed resources and deliver them      *
* to the warehouse where you will store      *
* then until the ship is completed. Then     *
* You will also need to build the ship,      *
* load the ship and then set sail for        *
* the promised land. You will first need     *
* to find the resources and manufacture the  *
* tools need to build the ship.              *
                                             *
* Good luck and have fun in this adventure.  *
*                                            *
**********************************************

*** displayStartProgram() function called ***
BUILD SUCCESSFUL (total time: 0 seconds)
```

Implement the `displayStartProgramView()` *function*

We now need to implement the `displayStartProgramView()` function. This is primary function that is responsible for implementing the basic user request cycle for the Start Program end user story.

- Prompt for and get the input
- Do the action
- Determine and display the next view implementation of the view. Here is the algorithm for the `startProgram()` function showing the basic steps that need to be performed.

The display function for all most all views use a similar algorithm. You can use the same algorithm as a template for nearly every View Layer class you create. Here is the basic algorithm:

Basic template for displaying a view:

```
displayView(): void
BEGIN
  do
    Prompt for and get the input value
    if (value == "Q") then
      return

    do requested action and display the next view

  while the view is not done
END
```

The three basic steps steps of the user request cycle are repeated over and over in a do-while loop until end user enters valid input and the action is completed successfully or the end user explicitly enters a value (e.g., "Q") to quit.

Open the `StartProgramView` class and scroll down and locate the `displayStartProgramView()` function. Delete the line that prints out a message indicating that this function was called and copy the basic algorithm from the template above into the function as a comment.

```
public void displayStartProgramView() {
    /*
    do
        prompt for and get playersName
        if playersName == 'Q' then
            return

        do  requested action and display next view

    while the view is not done
    */

}
```

The steps in the algorithm to prompting for and getting the players name and doing the associated action are somewhat complex so we again use Divide and Conquer to implement these task. Rather than writing all of the code in this one function, we will create separate functions to perform each of these task. This makes the display function simpler to understand, develop, test and maintain. It also allows to postpone worrying about the logic associated with these task. These functions will be developed in a later step.

> **Good Practice:**
>
> It is a good idea to create a separate function anytime you have a group of statements that are responsible for accomplishing some subtask in your code and then call that function. This makes your code more readable, reusable, easier to change and debug.

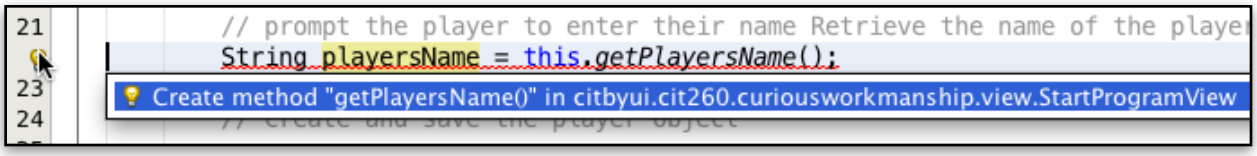
Translate each of the steps in the algorithm into it's corresponding Java code line by line as shown below.

```
30      /**
31       * displays the start program view
32       */
33      public void displayStartProgramView() {
34
36          boolean done = false; // set flag to not done
37          do {
            // prompt for and get players name
            String playersName = this.getPlayersName();
39          if (playersName.toUpperCase().equals("Q")) // user wants to quit
40              return; // exit the game
41
42          // do the requested action and display the next view
            done = this.doAction(playersName);
44
45          } while (!done);
46
47      }
```

First, we create a local variable called `done` to act as a signal flag to indicate when the view has completed. It is set initially to `false` to indicate that the view has not yet completed. We then used a do-while loop to repeat the end user request cycle until the view has completed

successfully. Next, we call the `getPlayerName()` function to prompt for and get the users input. The value returned is saved in the local variable, `playersName`. We then check to see if the value entered by the end user is equal to the literal value, “Q” (for quit). If true, we execute the `return` statement to exit the function; otherwise, controls skips down and calls the `doAction()` function. This function is responsible for creating and saving the Player object and for determining and displaying the next view.

Notice that that `getPlayersName()` and `doAction()` functions are underlined in red indicating that they have not been implemented yet. Click on the lightbulb and select the suggestion to create a stub function for the call to the `getPlayersName()` function in this class.



Repeat these steps to create the `doAction()` function. Your code should now look like this.

```
/**
 * displays the start program view
 */
public void displayStartProgramView() {

    boolean done = false; // set flag to not done
    do {
        // prompt for and get players name
        String playersName = this.getPlayersName();
        if (playersName.toUpperCase().equals("Q")) // user wants to quit
            return; // exit the game

        // do the requested action and display the next view
        done = this.doAction(playersName);

    } while (!done);

}

private String getPlayersName() {
    throw new UnsupportedOperationException("Not supported yet.");
}

private boolean doAction(String playersName) {
    throw new UnsupportedOperationException("Not supported yet.");
}
```

Notice that NetBeans automatically inserted the `throw new UnsupportedOperationException` into both the `getPlayersName()` and `doAction()` functions. This statement forces your program to terminate when the function is called. NetBeans did this to force you as the developer to implement this function. Unfortunately, it also prevents us from being able to test the `displayStartProgramView()` function. To overcome this limitation for now it is a good idea implement these two functions as stub functions.

Stub functions

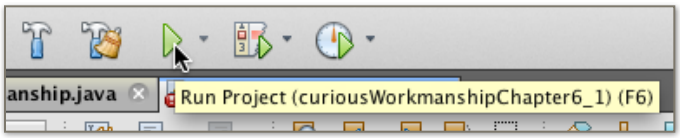
A stub functions acts as a temporary place holder. It contains only the minimal amount of code needed to test the that function is being called properly. Stub functions allow us to practice step wise incremental development where we write a little bit of code and then run the program to test and verify that the program written is working correctly for the code currently being worked on. Once that code is working as expected, we move on and then devote our full attention to the detailed implementation of the stub function without worrying about the code we already developed and tested earlier This makes it easier to debug where problems are occurring in our code.

Normally a stub function prints out the name of the function being called and then returns a dummy success value. Turn the `getPlayersName()` and `doAction()` functions into stub functions by replacing the `throw new UnsupportedOperationException` statements with the code below.

```
private String getPlayersName() {
    System.out.println("\n*** getPlayersName() called ***");
    return "Joe";
}

private boolean doAction(String playersName) {
    System.out.println("\n*** doAction() called ***");
    return true;
}
```

Run and test your program by selecting the name of your project in the Projects window and selecting the run button. Your results should be similar to this. Notice the two messages at the end indicating that our `getPlayerName()` and `doActions()` functions were called successfully.




```
*********************************************
*                                           *
* This is the game of Curious Workmanship   *
* In this game you will help Nephi build a  *
* shiop of curious workmanship to travel to *
* the promised land.                        *
*                                           *
* Good luck and have fun in this adventure. *
*                                           *
*********************************************

*** getPlayersName() function called ***

*** doAction() function called ***
BUILD SUCCESSFUL (total time: 0 seconds)
```

Implement the `getPlayersName()` *function*

Lets move on and now focus on the development the implementation for the `getPlayersName()` stub function called in the `displayStartProgramView()` function.

```
/**
 * displays the start program view
 */
public void displayStartProgramView() {

    boolean done = false; // set flag to not done
    do {
        // prompt for and get players name
        String playersName = this.getPlayersName();
```

This function is responsible for prompting and getting the players name. Here is a definition of what the function is supposed to do.

Task: Prompt the user to enter their name and then get the name

Inputs: name of player

Outputs: A player object

Validation: The name of the value entered must not be blank.

This is the test matrix for the function.

getPlayersName Test Matrix		
	Test Cases	
	Valid	Invalid
	1	2
Inputs		
name	"Fred Flintsone"	" "
Outputs		
value	"Fred Flintsone"	
Error		The value can not be blank.

Template for prompting for and getting input

Prompting for and getting input from an end user is done frequently in text based programs. Here is a generic template for an algorithm to prompt for and get input from and end user. Use this same template anytime you need to prompt for and get user input.

A template for getting user input:

```
getInput(): value
BEGIN
  WHILE a valid value has not been entered
    DISPLAY a message prompting the user to enter a value
    GET the value entered from keyboard
    Trim front and trailing blanks off of the value

    IF the length of the value is blank THEN
      DISPLAY "Invalid value: The value cannot be blank"
      CONTINUE
    ENDIF

    BREAK
  ENDWHILE

  RETURN value
END
```

In the algorithm above, a `WHILE` statement is used to repeat the block of code while a valid value has not been entered. A `WHILE` statement is used because we do not know how many times we will need to repeat the block of code. Every time the block is repeated, a message is displayed to prompt the end user to enter the desired value. The value entered by the end user is then retrieved from the keyboard. Any excess blanks are then trimmed off of the front and back of the value. We then check to see if an invalid value was entered. If an invalid value is detected, an error message is display and control immediately continues with the next iteration of the loop. If a valid value is entered then we break out of the repetition and then return the value entered.

Let's use this basic template and modify it to fit the needs of the `getPlayersName()` function. First scroll down to the getPlayersName() function at the bottom of the class.

We are now going to fully implement the function so we first need to delete the the code we added to make this a the stub.

```
        private String getPlayersName() {
Delete      System.out.println("\n*** getPlayersName() called ***");
            return "Joe";
        }
```

Then type in the algorithm to prompt for and get end user input.

```
private String getPlayersName() {
    /*
    WHILE valid value has not be entered
        DISPLAY promptMessage
        GET the value entered from keyboard
        Trim front and trailing blanks off of the name

        IF the length of the value is blank THEN
            DISPLAY "Invalid value: The value can not be blank"
            CONTINUE
        ENDIF

        BREAK

    ENDWHILE
    RETURN name
    */
}
```

The only changes we made to the template was to change the condition that checks for an invalid input value. The name is invalid if it's length is less than one character long (i.e., blank).

Translate each line of the algorithm into it's corresponding Java statements one at a time from top to bottom. Leave the comments that you typed in to document our code. This is a good programming practice.

Good Programming Practice:
Comment your code so that others can understand the intent of your algorithm. This allows you and other to easily understand what your function is doing when changes have to be made to the code much later.

```java
private String getPlayersName() {

    Scanner keyboard = new Scanner(System.in); // get infile for keyboard
    String value = ""; // value to be returned
    boolean valid = false; // initialize to not valid

    while (!valid) { // loop while an invalid value is enter
        System.out.println("\n" + this.promptMessage);

        value = keyboard.nextLine(); // get next line typed on keyboard
        value = value.trim(); // trim off leading and trailing blanks

        if (value.length() < 1) { // value is blank
            System.out.println("\nInvalid value: value can not be blank");
            continue;
        }

        break;  // end the loop
    }

    return value; // return the value entered
}
```

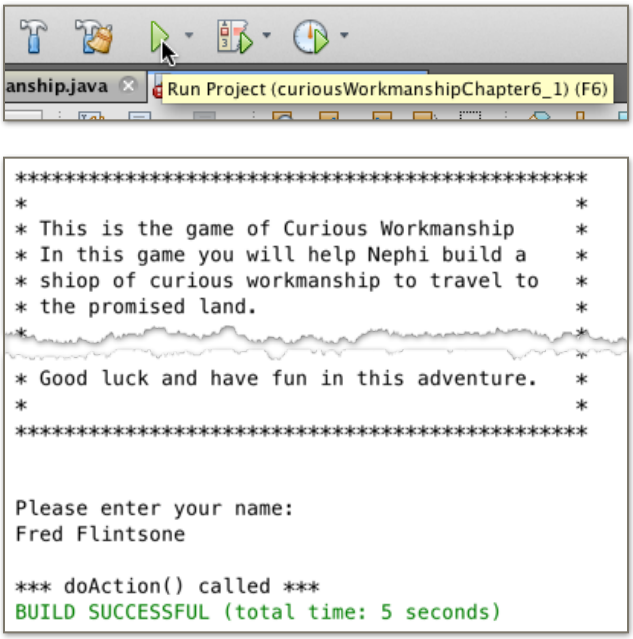
The function starts by defining three variables. A new `Scanner` object is then created and assigned to the `keyboard` variable. A `Scanner` object models an input file stream. An input file stream connects your program to a specific input file on your system. The `System.in` variable passed to the `Scanner()` function indicates that the input file stream is to be connected to the operating system's default input device. This is normally the keyboard. The `valid` variable is boolean value used signal when an valid name has been entered. We initialized the value of this variable to `false` to indicate that a valid value has not been yet entered. The `value` variable will store the value entered by the end user.

The Java `while` repetition is used to loop through and repeat the block of statements following it until a valid value is entered. The block of statements are executed while the value of the `valid` variable is not `true`. The first statement in the block calls the `println()` function to display the prompt message to the console. We set the value of `this.promptMessage` set earlier in the StartProgramView() constructor function. The`Scanner` object's `nextLine()` function is then called to read the next line of text from the keyboard. The value returned is always a `String`. It is then assigned to the `value` variable. The `String` `trim()`

function is used to remove any leading and trailing blanks from the value. Next, we check to see if the value entered is less than one character long. This validates that the end user did not enter a blank value. The `String` class's `length()` function is used to get the length of the `value` entered. If the value is less than one character long, an error message is displayed and the `continue` statement is executed. The `continue` statement causes control to immediately jump back up to the top of the loop and repeat the block of code is repeated until a valid value is entered. When valid value is entered, the `break` statement is executed. The `break` statement ends the loop and exits the repetition statement. The valid value entered is then returned. Control then returns back to the statement in the `displayStartProgramView()` function where the `getPlayersName()` function was called.

Run your program and enter the Players name to see if it is working correctly. Test entering input values defined in the test matrix for both test cases 1 and 2 to prove that your function is working properly. The results from running test 1 for your program should be similar the the output shown.

getPlayersName Test Matrix		
	Test Cases	
	Valid	Invalid
	1	2
Inputs		
name	"Fred Flintsone"	" "
Outputs		
value	"Fred Flintsone"	
Error		The value can not be blank.



Implement the `doAction()` function

Next, implement the `doAction()` function called at the end of the `displayStartProgramView()` function.

```
/**
 * displays the start program view
 */
public void displayStartProgramView() {

    boolean done = false; // set flag to not done
    do {
        // prompt for and get players name
        String playersName = this.getPlayersName();
        if (playersName.toUpperCase().equals("Q")) // user wants to quit
            return; // exit the game

        // do the requested action and display the next view
        done = this.doAction(playersName);

    } while (!done);

}
```

The `doAction()` function has the responsibility for performing the last two steps of the end user request cycle (do the requested action and displaying the next view).

Here is the algorithm for the `doAction()` function.

```
doAction(playersName): boolean
BEGIN
   if the length of the playersName < 2 then
      display "Invalid name: The name must be > 1 character"
      return false

   create Player with specified name
   if unsuccessful then
      display "Invalid name: The name is too short"
      return false

   display customized welcome message
   display mainMenuView
   return true
END
```

The input to the function is the player's name entered by the end user. The function checks for an invalid player's name is less than the required length. If invalid it displays an error message and returns a false value indicating that the function failed. If a valid name is entered, a create a new `Player` object with the specified players name is create. If an error occurred creating the `Player` object, an error message is displayed and a false value is returned to indicate that the `doAction()` failed. If no error occurred, a customized welcome message is printed and and the main menu view displayed.

Scroll down to the `doAction()` function in your program, delete the `throw new UnsupportedOperationException` statement and copy the algorithm above into your function as comments.

```
private boolean doAction(String playersName) {

    // if the length of the playersName < 2 then
        // display "Invalid name: The name must be > 1 character"
        // return false

    // create Player with specified name
    // if unsuccessful then
        //  display "Invalid name: The name is too short"
        //  return false

    // display customized welcome message
    // display mainMenuView
    // return true

}
```

Translate each of the statements into Java.

```
private boolean doAction(String playersName) {

    if (playersName.length() < 2) {
        System.out.println("\nInvalid players name: "
                + "The name must be greater than one character in length");
        return false;
    }
```

We call the `length()` string function in the condition of the if statement to get the length of the value stored in the `playersName` variable. If the length is less than two characters we print the error message to the console and return `false` to exit the function.

Creating a `Player` object is a function of the control layer because we are saving model layer data. We we call a new control layer function called `createPlayer()` in the `GameControl` class to create the.

Remember:

Remember that all functions that deal with making calculations, performing actions, make decisions and that get and save data in the Model Layer are to be implemented in the Control Layer.

This function will return the `Player` object created and assign it to the player variable. If the value returned is `null` an error occurred, an error message is printed to the console and a false value is returned to indicate that the function failed.

```
private boolean doAction(String playersName) {

    if (playersName.length() < 2) {
        System.out.println("\nInvalid players name: "
                + "The name must be greater than one character in length");
        return false;
    }

    // call createPlayer() control function
    Player player = GameControl.createPlayer(playersName);

    if (player == null) { // if unsuccessful
        System.out.println("\nError creating the player.");
        return false;
    }
```

Notice that the `Player` class and the reference to the `GameControl` classes are underlined in red. This is because Java can not find a definition of these these two classes in this class. Select the lightbulb and click on the suggestion to import the `Player` class.

```
63          // call createPlayer() control function
            Player player = GameControl.createPlayer(playersName);
65
            Add import for citbyui.cit260.curiousworkmanship.model.Player
```

The `GameControl` class name is also underlined in red and needs to be imported. Select the lightbulb and click on the suggestion to import the `GameControl` class. **Note**: If you have not created the `GameControl` class yet, you will need to create the `GameControl` class in your project's "control" package before you can import it.

```
65        // call createPlayer() control function
          Player player = GameControl.createPlayer(playersName);
67
Add import for citbyui.cit260.curiousworkmanship.control.GameControl
```

The `createPlayer()` function is now underlined in red because this function can not be found in the `GameControl` class because it has not been implemented yet. Click on the lightbulb and select the suggestion to create this function in the `GameControl` class.

```
66        // call createPlayer() control function
          Player player = GameControl.createPlayer(playersName);
Create method "createPlayer(java.lang.String)" in citbyui.cit260.curiousworkmanship.control.GameControl
```

Open the `GameControl` class in your Control Layer package and locate the createPlayer function. Turn it into a stub function so that we can run and test that the `createPlayer()` function is being called correctly in the `doAction()` function.

```
public static Player createPlayer(String playersName) {
    System.out.println("\n*** createPlayer() function called ***");
    return new Player();
}
```

The next steps in the `doAction()` function algorithm display a customized welcome message and display the main menu needs to be implemented.

```
private boolean doAction(String playersName) {

    // if the length of the playersName < 2 then
        // display "Invalid name: The name must be > 1 character"
        // return false

    // create Player with specified name
    // if unsuccessful then
        //  display "Invalid name: The name is too short"
        //  return false

    // display customized welcome message
    // display mainMenuView
    // return true

}
```

This will require several statements so we will use divide and conquer technique and create a new function called `displayNextView()` to implement this part of the algorithm. Open up the `StartProgramView` class again and scroll back down to the `doAction()` function and type in the statement to call a `displayNextView()` function.

```
private boolean doAction(String playersName) {

    if (playersName.length() < 2) {
        System.out.println("\nInvalid players name: "
                + "The name must be greater than one character in length");
        return false;
    }

    // call createPlayer() control function
    Player player = GameControl.createPlayer(playersName);

    if (player == null) { // if unsuccessful
        System.out.println("\nError creating the player.");
        return false;
    }

    // display next view
    this.displayNextView();
```

Noticed that we qualified the function with the keyword, `this`. The `this` keyword tells Java to look for this function in "this" class.

The call to the `displayNextView()` function is underlined in red because this symbol can not be found in "this" class. Click on the lightbulb and select the suggestion to create the `displayNextView()` function.

```
121         // display next view
            this.displayNextView(player);
1   Create method "displayNextView(citbyui.cit260.curiousworkmanship.model.Player)"
```

Scroll down to the `displayNextView()` function in this class and turn it into a stub function by replacing the `UnsupportedOperationException` statement with a line that print out a message indicating that the function has been called.

```
private void displayNextView(Player player) {
    System.out.println("\n*** displayNextView() called ***");
}
```

Scroll back up to the `doAction()` function and implement the last statement in the algorithm that returns a `true` value.

```
private boolean doAction(String playersName) {

    if (playersName.length() < 2) {
        System.out.println("\nInvalid players name: "
                + "The name must be greater than one character in length");
        return false;
    }

    // call createPlayer() control function
    Player player = GameControl.createPlayer(playersName);

    if (player == null) { // if unsuccessful
        System.out.println("\nError creating the player.");
        return false;
    }

    // display next view
    this.displayNextView(player);

    return true; // success !
}
```

A `true` value returned from the `doAction()` function will end the view. A `false` value will force the view to be redisplayed.

Run and test your program now. You results should similar.

```
***********************************************
*                                             *
* This is the game of Curious Workmanship     *
* In this game you will help Nephi build a    *
* shiop of curious workmanship to travel to   *
* the promised land.                          *
*                                             *
* Good luck and have fun in this adventure.   *
*                                             *
***********************************************


Please enter your name:
Fred Flintstone

*** createPlayer() function called ***

*** displayNextView() called ***
BUILD SUCCESSFUL (total time: 5 seconds)
```

Implementing the `CreatePlayer()` *function*

We need to implement the `createPlayer()` function in the `GameControl` class. This function will create and return a `Player` object instance. Here is the test matrix and algorithm for the function:

createPlayer Test Matrix		
	Test Cases	
	Valid	Invalid
	1	2
Inputs		
name	"Fred Flintsone"	null
Outputs		
playersName	"Fred Flintsone"	null
Error		The name is nulll

There is only one invalid test case. We need to check to make sure that the value of the `name` of input parameter contains a value and is not `null`. The `null` keyword means that the variable does not contain a value.

Here is the algorithm for the function.

```
createPlayer(name): Player
BEGIN
   IF (name == null ) then
      return null

   Create Player object
   Set the name in the player object
   Save the Player as a global variable
END
```

The first step of any algorithm is to check for invalid input parameters. In this case, the value of the `name` input parameter variable is invalid is `null`. A `null` value means that no value was passed to the name variable when the function is called. If this condition evaluates to `true`, the function exits and a `null` value (meaning nothing) is returned to signal that an error occurred. A `null` value is returned instead of -1 because the function

signature requires that the data type of the returned value must be a reference to a `Player` object. If a valid value was passed to the `name` input parameter, a new `Player` object instance is created, the `Player` object's `name` attribute is set to the value of the `name` input parameter variable, and the the Player object is saved in a global variable that it can be accessed directly from any class in the program.

You need to open the `GameControl` class and locate the `createPlayer()` function to implement this function.

```
public static Player createPlayer(String playersName) {
    System.out.println("\n*** createPlayer() function called ***");
    return new Player();
}
```

Replace the statement entered earlier when you made this a stub function. Delete those lines and implementation the full function as shown below.

```
24    public class GameControl {
25
26        public static Player createPlayer(String name) {
27
28            if (name == null) {
29                return null;
30            }
31
32            Player player = new Player();
33            player.setName(name);
34
            CuriousWorkmanship.setPlayer(player); // save the player
36
37            return player;
38        }
```

First we check to see if name `name` input parameter is `null` invalid input parThe `if` statement checks to see if the value of the. A new `Player` object is created and assigned to a local variable of type `Player`. The `Player` object's `setName()` function is then called to assign a value to the name attribute of the `Player` object. The `setPlayer()` function is then called to save the new `Player` object in a global static variable in `CuriousWorkmanship` class. We know the `setPlayer()` function is static because it is qualified with a class name instead of a variable that references a specific object instance. Finally, we return the new `Player` object created.

Notice that the reference to the `CuriousWorkmanship` class is underlined in red. This error is occurring because the `CuriousWorkmanship` class has not been imported into this file. Select the lightbulb and click on the suggestion to add the import statement for this class.

The `setPlayer()` function is now underlined in red and the `cannot find symbol` error occurs because Java can not find the `setPlayer()` function in the `CuriousWorkmanship` class. This makes sense because we have not yet defined a global static variable and it's corresponding getter and setter functions in the `CuriousWorkmanship` class.

```
35
        CuriousWorkmanship.setPlayer(player); // save the player
```

To solve this problem we need to add a `private static` class variable called `player` to hold a reference to the current player object. We also need to add a static class variable called `currentGame`. This variable will hold a reference to the current game object instance. Open the class with the `main()` function in your program (e.g., `CuriousWorkmanship` in the example game). This is the class that has the same name as your program. Add static class variables to store a reference to the current game and the player of the game.

```
public class CuriousWorkmanship {

    private static Game currentGame = null;
    private static Player player = null;

    public static void main(String[] args) {
```

Now create `public static` getter and setter functions for these two variables at the bottom of the class (Hint: Use the same **Insert Code** command that automatically created the "getter and setter" functions that used to create the Java Bean objects in the Model Layer).

```
public static Game getCurrentGame() {
    return currentGame;
}

public static void setCurrentGame(Game currentGame) {
    CuriousWorkmanship.currentGame = currentGame;
}

public static Player getPlayer() {
    return player;
}

public static void setPlayer(Player player) {
    CuriousWorkmanship.player = player;
}
```

Now go back down to the `createPlayer()` function in the `GameControl` class. The error should now be gone.

```
public static Player createPlayer(String name) {

    if (name == null) {
        return null;
    }

    Player player = new Player();
    player.setName(name);

    CuriousWorkmanship.setPlayer(player); // save the player

    return player;
}
```

Implement the `displayNextView()` function

Now we need to implement the `displayNextView()` stub function we created earlier. Here is the algorithm for the displayNextView() function.

```
displayNextView(player): Player
BEGIN
   Print a customized welcome message

   Create the MainMenuView
   display the MainMenuVIew
END
```

First, a customized welcome message will be displayed with the players name. Then we need to display the main menu. The main menu is an entirely different view from the `StartProgramView`. To display the view we first need to create a new MainMenuView object and then call it's displayMainMenu function.

Open the `StartProgramView` class again and scroll down to the `displayNextView()` stub function created earlier.

```
private void displayNextView(Player player) {
    System.out.println("\n*** displayNextView() called ***");
}
```

Type in the following code to implement the `displayNextView()` function.

```
private void displayNextView(Player player) {

    // display a custom welcome message
    System.out.println("\n==============================================="
                       + "\n Welcome to the game " + player.getName()
                       + "\n We hope you have a lot of fun!"
                       + "\n==============================================="
                       );

    // Create MainMenuView object
    MainMenuView mainMenuView = new MainMenuView()

    // Display the main menu view
    mainMenuView.displayMainMenuView();
}
```

First, we print the custom welcome menu and insert the player's name in the message by calling the player object's `getName()` function.

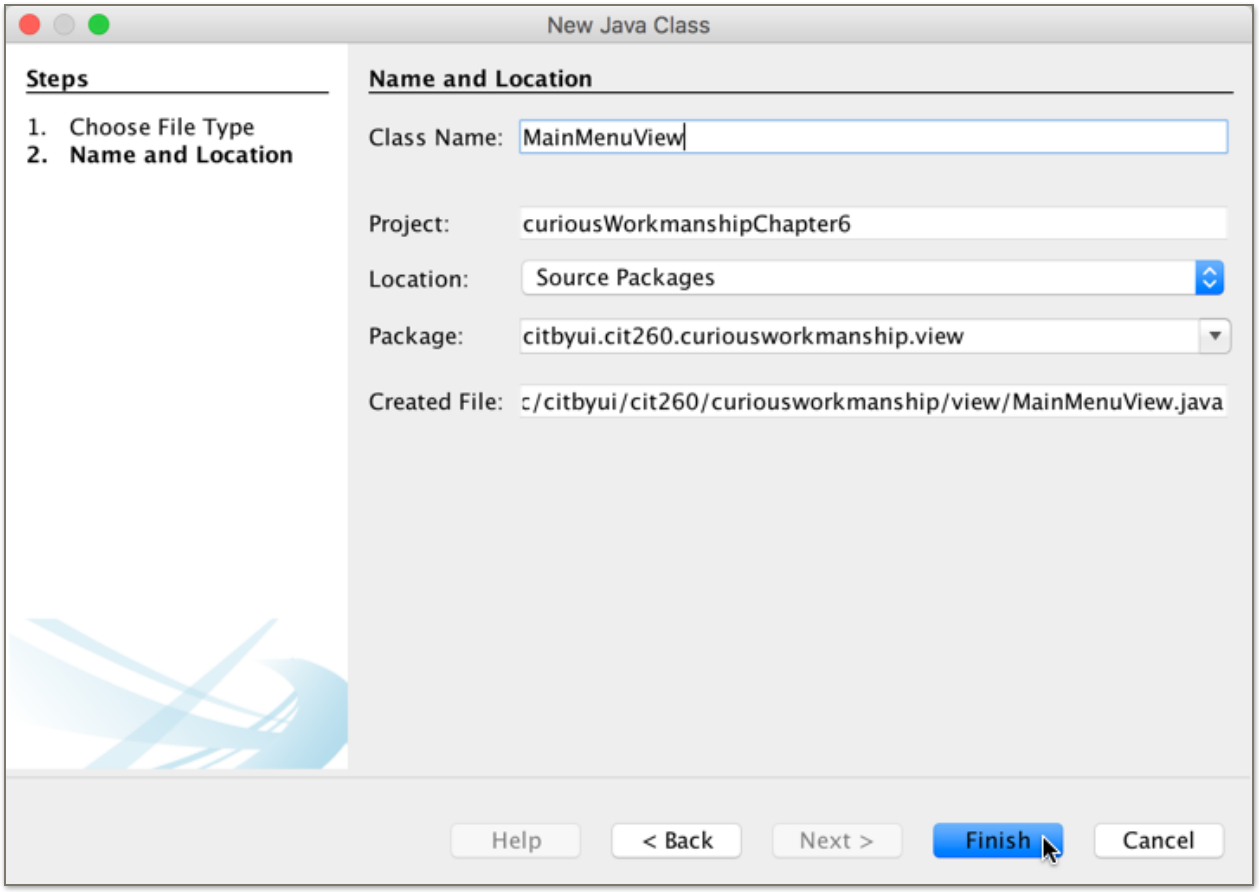
The main menu is an entirely different view that will be implemented in the `MainMenuView` class. A new object instance of the `MainMenuView` class must be first created and saved in the `mainMenuView` variable. Then we called it's `displayMainMenuView()` function to display the main menu.

The MainMenuView class is underlined because this symbol has can not it has not been implemented yet and can not be found. We need to create this class now. Select the View Layer package.

Create the MainMenuView class in the View Layer package.

We will postpone the full implementation of the `MainMenuView` class to a later step.

Back in the `displayNextView()` function the call to the `displayMainMenuView()` function is also underlined in red because this function has not been defined yet.

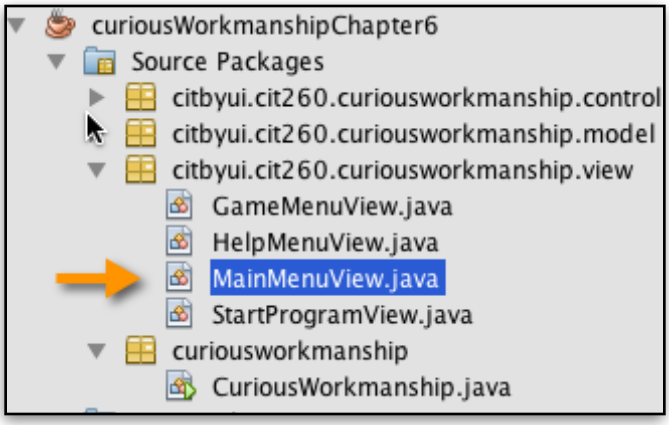


Let's create the create the `displayMainMenuView()` function in the `MainMenuView` class so that we can call and test this function. Click on the lightbulb and select the suggestion to create the `displayMainMenuView()` function.

```
93          // Display the main menu view
            mainMenuView.displayMainMenuView();
95
96
97      }
```

Create method "displayMainMenuView()" in citbyui.cit260.curiousworkmanship.view.MainMenuView

Open the `MainMenuView` class and scroll down and find the `displayMainMenuView()` function.




```
public class MainMenuView
{

    void displayMainMenuView() {
        throw new UnsupportedOperationException("Not supported yet.");
    }
```

Let's turn this into a stub function by replacing the `throw new UnsupportedOperationException` statement with the a statement that prints a message indicating that we called the function.

```
void displayMainMenuView() {
    System.out.println("\n*** displayMenu() function called ***");
}
```

Run and test your program. Your results should look similar to figure below.

```
***********************************************
*                                             *
* This is the game of Curious Workmanship     *
* In this game you will help Nephi build a    *
* shiop of curious workmanship to travel to   *
* the promised land.                          *
*                                             *
* Good luck and have fun in this adventure.   *
*                                             *
***********************************************


Please enter your name:
Fred Flintstone

=============================================
 Welcome to the game Fred Flintstone
 We hope you have a lot of fun!
=============================================

*** displayMainMenuView() function called ***
BUILD SUCCESSFUL (total time: 7 seconds)
```

Make sure you save all of your changes. It also is a good time to commit all of your changes to your local repository. Then move on to the next step where you will create the MainMenuView class for your Main Menu end user story.

Step 2 - Implementing the `MainMenuView` class

We will use the same the Divide and Conquer technique we used earlier to create the the functions in the `MainMenuView` class.

The last two statements in the `displayNextView()` function create an instance of the `MainMenuView` class and then called the `displayMainMenuView()` function to display the main menu view.

```
private void displayNextView(Player player) {

    // display a custom welcome message
    System.out.println("\n=============================================="
                       + "\n Welcome to the game " + player.getName()
                       + "\n We hope you have a lot of fun!"
                       + "\n==============================================="
                       );

    // Create MainMenuView object
    MainMenuView mainMenuView = new MainMenuView();

    // Display the main menu view
    mainMenuView.displayMainMenuView();
}
```

We need to first create the default constructor function for the `MainMenuView` class so that a new object instance can be created. Next we need to create the `displayMainMenuView()` function in the `MainMenuView` class so the main menu view can be displayed.

Implementing the `MainMenuView()` constructor function.

The main job of a constructor function is to initialize any class instance variables used by the functions in the class. The only class instance variable that needs to be defined and initialized in the `MainMenuView` class is the `menu` variable. This will hold the default menu to be displayed.

Open the `MainMenuView` class and create a private class instance variable.

```
public class MainMenuView
{
    private String menu;
```

Now create the default constructor function for the class. Initialize the `menu` class instance variable with the text to be displayed on your main menu. It should look similar to this.

```
public MainMenuView() {
    this.menu = "\n"
              + "\n----------------------------------------"
              + "\n| Main Menu                            |"
              + "\n----------------------------------------"
              + "\nN - Start new game"
              + "\nG - Get and start saved game"
              + "\nH - Get help on how to play the game"
              + "\nS - Save game"
              + "\nQ - Quit"
              + "\n----------------------------------------";
}
```

After the constructor function is called to create a new object instance, the classes display function is called to display the view. The display function is responsible for the overall behavior of the view. Implement the `displayMainMenuView()` function in the `MainMenuView` class next.

Implement the `displayMainMenuView()` function

After studying the end user story we came up with the following problem definition for the `displayMainMenuView()` function.

Main Menu

Display the following menu.

```
N - Start new game
G - Get and start a saved game
H - Get help on how to play the game
S - Save game
Q - Quit
```

The end user (player) enters the selected item. The computer then displays the selected scene. The program ends when the Quit menu item is selected.

Task: Display the main menu, get the users selection and perform the selected action. Continue displaying the menu until the player chooses to exit the program.

Inputs: A menu item

Outputs: The next view associated with the selected action

Validation: The menu item must be either “G”, “H”, “S” or “Q”

Here is the test matrix we developed for the `displayMainMenuView()` function.

displayMainMenuView() Test Matrix							
	Test Cases	Test Cases					
		Valid				Invalid	
	1	1	2	3	4	5	6
Inputs							
selection	"N"	"G"	"h"	"s"	"Q"	" "	"z"
Outputs							
	Display GameMenu View	Display GetGame View	HelpMenu Menu	SaveGame Game view	Exit the program	Display Invalid Selection error	Display Invalid Selection error

We use the same basic display algorithm template used to implement the `displayStartProgramView()` function to implement the `displayMainMenuView()` function.

Basic template for displaying a view:

```
displayView(): void
BEGIN
  do
    Prompt for and get the input value/s
    if (value == "Q") then
      exit

    do the action and display the next view

  while the view is not done
END
```

The code to implement function to `displayMainMenuView()` function should be very similar to that of `displayStartProgramView()` functions because they are using the same basic algorithm.

Open the `MainMenuView` class created earlier and scroll down to the `displayMainMenuView()` stub function.

```
public class MainMenuView
{
    public void displayMainMenuView() {
        System.out.println("\n*** displayMainMenuView() function called ***");
    }
```

Delete the statement that prints out the message indicating that the function was called. Copy the algorithm above into the `displayMenu()` function. Translate each of the statements in the algorithm into Java statements. The result should be as follows.

```java
40          /**
41           * displays the start program view
42           */
43          public void displayMainMenuView() {
44
            boolean done = false; // set flag to not done
46          do {
47              // prompt for and get players name
                String menuOption = this.getMenuOption();
49              if (menuOption.toUpperCase().equals("Q")) // user wants to quit
50                  return; // exit the game
51
52              // do the requested action and display the next view
                done = this.doAction(menuOption);
54
55          } while (!done);
56
57      }
```

In the function, the done variable serves as a flag to indicate when the view has been completed. The do-while repetition will continue until either the users enters a “Q” to quit or when the `doAction()` function returns a `true` value. The `getMenuOption()` function is responsible for prompting f and getting the menu option entered by the end user. the `doAction()` function is responsible for selecting and performing the action associated with the menu option entered.

The statements calling the `getMenuOption()` and `doAction()` functions have errors. Hovering your cursor over the lightbulb reveals that the `undefined symbol` error. These functions have not been created yet in this function. Click on the lightbulb for each function and select the suggestion to create the function in this class. Then replace the `throw new UnsupportedOperationException` statement with a statement that prints a message indicating that the function was called successfully and add a statement to return a valid value. This is what your code should look like once you have made those changes.

```java
/**
 * displays the start program view
 */
public void displayMainMenuView() {

    boolean done = false; // set flag to not done
    do {
        // prompt for and get players name
        String menuOption = this.getMenuOption();
        if (menuOption.toUpperCase().equals("Q")) // user wants to quit
            return; // exit the game

        // do the requested action and display the next view
        done = this.doAction(menuOption);

    } while (!done);

}

private String getMenuOption() {
    System.out.println("\n*** getMenuOption() function called ***");
    return "N";
}

private boolean doAction(String menuOption) {
    System.out.println("\n*** doAction() function called ***)");
    return true;
}
```

Now run and test your program.

Develop the `getMenuOption()` function

Implement the `getMenuOption()` stub function created in the last step. This function is very similar to the create your `getPlayersName()` function developed in the StartProgramView class earlier.

Instructions:

Develop a test matrix for the `getMenuOption()` function defining unit test for all of the valid and invalid inputs.

Since this function needs to get input from the end user, we can use same template algorithm for getting user input described earlier.

A template for getting user input:

```
getInput(): value
BEGIN
  WHILE a valid value has not been entered
    DISPLAY a message prompting the user to enter a value
    GET the value entered from keyboard
    Trim front and trailing blanks off of the value

    IF the length of the value is blank THEN
      DISPLAY "Invalid value: The value cannot be blank"
      CONTINUE
    ENDIF

    BREAK
  ENDWHILE

  RETURN value
END
```

Modify the template to get the menu item entered by the end user. Then translate the algorithm into Java code. The code should be very similar to `getPlayerName()` function (Hint: Copy the `getPlayerName()` function created earlier and modify it to fit your algorithm. This will save you a lot of time).

Run your program and execute each of test cases defined in your test matrix to verify that your function is working as specified in the test matrix.

Develop the `doAction()` Function

Next we need to implement the `doAction()` function that is called in the `displayMenu()` function. Here is the definition of the `doAction()` function in the example program.

Task:	Determine which action is selected and then perform that action..
Inputs:	A menu item
Outputs:	boolean - false (redisplay the view)
Validation:	The menu item must be either “N”, “G”, “H”, “S” or

The test matrix is very similar to `displayMenu()` function except that we eliminated the test cases for “Q” because that case is covered in the `getMenuOption()` function and does not need to be retested here. Notice that a `false` value is returned for all of the test cases. Returning `false` will cause the menu to be redisplayed until the user enters “Q” to quit.

doAction() Test Matrix					
	Test Cases				
	Valid				Invalid
	1	1	2	3	6
Inputs					
selection	"N"	"G"	"h"	"s"	"z"
Outputs					
	FALSE	FALSE	FALSE	FALSE	FALSE

This function is doing a lot so we will again use Divide and Conquer to develop this function. Here is the basic algorithm for the `doAction()` function implemented in the example program.

```
doAction(choice): void
BEGIN
    convert choice to upper case
    SWITCH choice
        "N":    Start a new game
        "G":    Get and start a saved game
        "H":    Display the help menu
        "S":    Display the save game view
        DEFAULT: DISPLAY "Invalid selection"
    ENDSWITCH
    RETURN false
END
```

This function needs to determine which menu item was entered and then perform the associated action. A `switch` statement here is a good choice because we need to select from a list valid values. An error message is displayed when an invalid choice is entered (i.e., not N, G, H, or S).

Here is the is the function after it was translated it into Java code.

```java
public boolean doAction(String choice) {

    choice = choice.toUpperCase(); // convert choice to upper case

    switch (choice) {
        case "N": // create and start a new game
            this.startNewGame();
            break;
        case "G": // get and start an existing game
            this.startExistingGame();
            break;
        case "H": // display the help menu
            this.displayHelpMenu();
            break;
        case "S": // save the current game
            this.saveGame();
            break;
        default:
            System.out.println("\n*** Invalid selection *** Try again");
            break;
    }

    return false;
}
```

Each of the actions can be viewed as a separate subtask that may take several lines of code to implement. So we will implement separate functions for each action.

We do this by first creating stub functions in the `MainMenuView` class for each of the choices in the class.

```
private void startNewGame() {
    System.out.println("*** startNewGame function called ***");
}

private void startExistingGame() {
    System.out.println("*** startExistingGame function called ***");
}

private void saveGame() {
    System.out.println("*** startExistingGame function called ***");
}

private void displayHelpMenu() {
    System.out.println("*** displayHelpMenu function called ***");
}
```

We can now run and test the program to see if the Main Menu View is working correctly. The Main Menu view should display. Run all of the unit test cases defined in the `displayMainMenuView()` test matrix to prove that your function is working correctly for each menu option.

```
-----------------------------------------
| Main Menu                             |
-----------------------------------------
N - Start new game
G - Get and start saved game
H - Get help on how to play the game
S - Save game
Q - Quit
-----------------------------------------
n

 *** startNewGame() function called ***
```

Implement the stub functions

Now we need to implement each of these stub functions you just created for the `doAction()` function. Let's start with the `startNewGame()` function. It is responsible for creating a new game object and then display the game menu view. Here is the algorithm for the function.

```
startNewGame(): void
BEGIN
    Create a new Game

    Create a new Game Menu View
    Display the Game Menu
END
```

Moved down to the `startNewGame()` stub function in the `MainMenuView` class and typed the algorithm directly into the function as comments and then implemented each statement in the algorithm.

The first statement of the algorithm requires creating a new game. This is a relatively complex subtask so we decided to create and call a function called `createNewGame()` to implement this subtask.

```
120     private void startNewGame() {
121             // create a new game
            GameControl.createNewGame(CuriousWorkmanship.getPlayer());
123
```

This function deals with the play of the game so we will implement this function in the `GameControl` Control Layer class as a static function. Remember that `static` functions and variables are are global to all object instances of the class and do not belong to any specific object instance. Therefore, we qualify the function by it's class name instead of with a variable referencing and object of the class. Notice that we also qualified `getPlayer()` function with it's class name because it is also a static function.

The next two statements in the algorithm states that we need to create a new Game Menu View and then display it's menu. The game menu is defined in the Game Menu end user story so will need to create a new

class called `GameMenuView` to implement this view. Here is the code to implement these two statements.

```
120     private void startNewGame() {
121             // create a new game
            GameControl.createNewGame(CuriousWorkmanship.getPlayer());
123
124         // display the game menu
            GameMenuView gameMenu = new GameMenuView();
126         gameMenu.displayMenu();
127     }
```

Notice errors in this function. When we hover the cursor over the light bulbs we see the `cannot find the symbol` error for both references to the `GameControl` and `GameMenuView` classes.

```
117     }
118
119
120     ewGame() {
121     te a new game
            GameControl.createNewGame(CuriousWorkmanship.getPlayer());
123
124         // display the game menu
            GameMenuView gameMenu = new GameMenuView();
126         gameMenu.displayMenu();
127     }
```

```
cannot find symbol
  symbol:  variable GameControl
  location: class MainMenuView
----
(Alt-Enter shows hints)
```

We need to create these two classes. Create the `GameControl` class in the Control Layer package and the `GameMenuView` in the Model Layer packages, and then select the light bulb import these two classes.

```
Source Packages
  citbyui.cit260.curiousworkmanship.control
    GameControl.java
    InventoryControl.java
    MapControl.java
    ProgramControl.java
  citbyui.cit260.curiousworkmanship.model
  citbyui.cit260.curiousworkmanship.view
    GameMenuView.java
    MainMenuView.java
    StartProgramView.java
  curiousworkmanship
    CuriousWorkmanship.java
```

After importing the `GameControl` and `GameMenuView` classes into the `MainMenuView` class, we get the `cannot find symbol` errors again because we have not created the `createNewGame()` and `displayMenu()` functions yet.

```
121     private void startNewGame() {
122             // create a new game
            GameControl.createNewGame(CuriousWorkmanship.getPlayer());
124
125         // display the game menu
126         GameMenuView gameMenu = new GameMenuView();
            gameMenu.displayMenu();
128     }
```

Create stub functions for both of these function in the respective classes. The implementation of both of these functions will be done later.

```
public class GameControl {

    public static void createNewGame(Player player) {
        System.out.println("\n*** createNewGame stub function called ***");
    }
```

```
public class GameMenuView {

    void displayMenu() {
        System.out.println("\n*** displayMenu stub function called ***");
    }
```

Step 3 - Implement the `HelpMenuView` class

Here is the **Getting help** end user story in the example program. Notice that it is very similar to the **Main Menu** end user story except that there are different actions associated with the menu. Each selection will print out the help text associated with the selected action.

Getting help

The following menu is displayed.

```
G - What is the goal of the game?
M - How to move
E - Estimating the amount of resources
H - Harvesting resources
D - Delivering resources to warehouse
Q - Quit
```

The user selects one of the options and the appropriate help message is displayed. Return to either the either the main or game menu depending on which one was previous displayed before this menu.

Instructions:

1. Use the Main Menu view test matrix, algorithms and code you developed for `MainMenuView` in the previous step as a template to develop the view for the Help Menu end user story for your program. Modify the templates as needed to fit needs of your Help Menu end user story.

2. Then run your program and run each of the test cases defined in your test matrices. Fix any problems that you find and rerun you test cases.

Submit your assignment

1. Be sure to save all of your changes and commit your changes to the local repository. Pull from the remote GitHub repository and merge any changes into your local repository. If conflicts occur, resolve the conflicts and save your changes. Commit the changes to your local repository again. Finally, Push your code to the remote GitHub repository.

2. Submit your assignment and add a note with the name of each of your team members, the url of your repository and a list of the classes that your team modified.